Arquitecturas en Software

Comparaciones y Ventajas

Iniciemos un camino para mejorar la cultura del software. Aportemos valor juntos

¿Por qué Arquitectura de Software?

Una gran pregunta que ha impactado por más de 2 décadas a toda la comunidad de developers, **según fuentes de Stackshare se comparte que solo el 22% de los developers a nivel internacional conocen los términos esenciales de la Arquitectura de Software,** un porcentaje crítico que ha impactado a las organizaciones al momento de implementar software de alta disponibilidad.

Este porcentaje impacta desde la liberación de estudiantes de las universidades provenientes de carreras como de ingeniería de software, ingeniería de sistemas, tecnologías de la información, etc., ya que **los programas de estudio** de estas carreras **no dirigen conocimiento al diseño y aprovechamiento de arquitecturas de software,** esto nos lleva a que **la lógica de los developers egresados se oriente a producir código rápido, aprovechando liberarías ya construidas y liberando producto sin calidad, el cual impacta de una manera negativa a la industria del software.**

Es muy importante considerar que el aprovechamiento de las arquitecturas de software consiste en organizar los componentes de un sistema para que se ajusten mejor a los atributos de calidad deseados y esperados por el "Product Owner y sus usuarios", desde un entorno funcional hasta visual.

Sin embargo, para definir la arquitectura del software es importante contar con un caso de negocio, el cual atienda las necesidades estratégicas de la organización y sobre todo las capacidades del producto.

Uno de los grandes retos que tienen las organizaciones hoy en día, es saber reaccionar de manera inmediata ante los cambios en el mercado y en el comportamiento del consumidor, y aunque la mayoría de las compañías creen que la situación actual cambió o cambiará la forma de hacer software, solo el **21%** de las organizaciones **tiene los recursos y la experiencia necesaria para innovar.**

Todos queremos algo ("Un Software Extraordinario")

Desde el 2015 a la fecha (Junio 2021), la disponibilidad de la información y conocimientos en software se encuentra al alcance de todos, esto ha permitido que los usuarios cuenten con un mayor grado de demanda sobre sus deseos del software a construir. **Esto ha orientado a que los arquitectos de software previo al iniciar una valoración para definir un producto se encuentran con elementos como:**

1. El usuario desea que su producto sea rápido, confiable y de alta disponibilidad.
2. Los Project Manager / Scrum Masters se preocupan por que el producto salga en tiempo y con el presupuesto justo.
3. El "Product Owner" se preocupa por que el software a desarrollar cumpla con las necesidades del negocio.
4. El equipo de seguridad requiere que el software se encuentre blindado para toda capa de seguridad.
5. El equipo de Mantenimiento se preocupan por que el software sea bondadoso para entender y mantener.
6. No contamos con product backlog pero queremos que se parezca a Facebook, Amazon.com, Netflix, Kavak, Mercado Libre, etc.
7. Entre muchos otros más…

El cual encamina al Arquitecto de Software en hacer malabares con el software, es ahí donde aplica su momento creativo para definir estos atributos de calidad y patrones de arquitectura de software. Además de la identificación de la organización, si es una empresa pequeña o un corporativo, ya que esto le permite identificar atributos como:

- Performance
- Disponibilidad
- Usabilidad (UX / UI)
- Interoperabilidad
- Seguridad
- Portabilidad
- Escalamiento
- Despliegue (Stages)
- Patrones de Arquitectura

¿Por qué considerar un patrón de arquitectura?

Los patrones de arquitectura, nos orientan a organizar nuestro código, el cual en un futuro nos permitirá solucionar problemas de como estructurar la arquitectura de nuestro software y ofrecer un producto de alta disponibilidad.

Visualicemos que necesitamos conocer las reglas que nos ofrece nuestra arquitectura de software para realizar la construcción de nuestro producto, esto es lo que hace posible la interacción entre distintas partes del sistema, ya que podemos realizar una buena arquitectura logrando objetivos como:

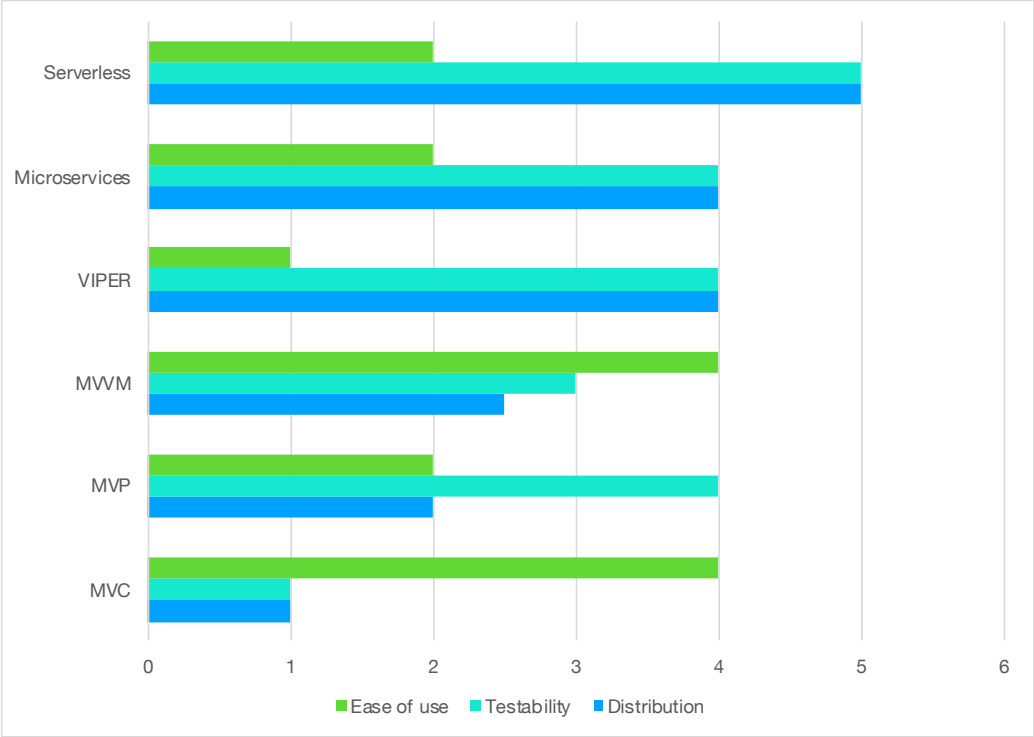
- Crear una base solida de nuestros sistemas
- Definir una plataforma que sea escalable
- Garantizar una mejor limpieza del código
- Reducción de código duplicado
- Mejorar la calidad y el rendimiento
- Integración limpia y disponible con otros sistemas y/o empresas.

Para lograr estos objetivos del software debemos de apoyarnos de distintos patrones arquitectónicos tales como:

- MVC
- MVP
- MVVM

- Viper
- Microservicios
- Serverless

Comparando los distintos modelos, podemos ver que nos ofrece cada uno de ellos a beneficio de la distribución de nuestro código, las pruebas o incluso a la facilidad de uso que requiere a la hora de interactuar con el, aquí una comparativa entre patrones de arquitecturas:



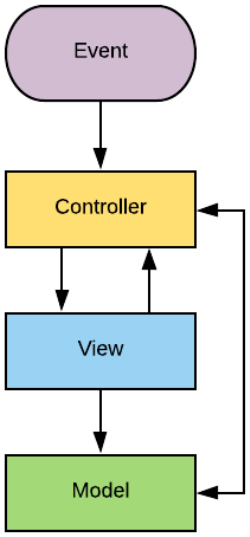
Fuente: XID – Digital Services

Hicimos una investigación para compartirte desde nuestro punto de vista como se comparan estos patrones de arquitectura, cuales son sus beneficios y como son aprovechados por algunas organizaciones.

Iniciaremos desde la parte fundamental que la gran de las organizaciones actualmente utilizan dentro de sus desarrollos de software:

MVC (Model – View - Controller)

Nace 1970 siendo el pionero de los demás patrones de arquitecturas. Inicialmente se definió como un patrón arquitectural, ya que colabora como una guía que comparte la manera de como organizar y estructurar los componentes del software, responsabilidades y relaciones entre ellos. Además, se considera un patrón de diseño en la capa de presentación, ya que define la organización de los componentes de presentación en softwares distribuidos



Su nombre, hace referencia a las capas de componentes en los que organizaremos nuestras aplicaciones bajo este paradigma, "Modelo – Vista – Controlador".

MVC propone, independientemente de las tecnologías, la separación de los componentes de una aplicación en tres capas:

- Modelo
- Vista
- Controlador

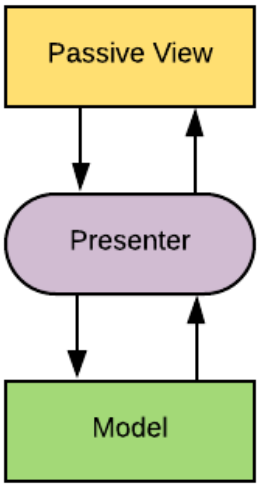
Describiendo el cómo se relacionarán entre ellos para mantener una estructura organizada, limpia y con un acoplamiento mínimo entre las distintas capas.

Debido a la antigüedad de esta arquitectura, no fue ideado para plataformas Web, aunque hoy en día se implementa bastante en ellas en empresas que inician su camino en el software. **Sin embargo, es una de las arquitecturas más complejas, costosas e ineficientes para los usuarios, ya que al momento de modificar algún componente del aplicativo podríamos dañar todo el sistema,** además de incrementar la demanda del servidor.

Un beneficio que hemos identificado es que parten en el entendimiento global a nivel de desarrollo, ya que todo desarrollador ha tenido la

oportunidad de participar en proyectos con esta arquitectura, por lo tanto, su inmersión en un proyecto es muy sencilla.

MVP (Model - View - Presenter)



El patrón de arquitectura MVP se deriva de una variante de MVC. La diferencia entre los dos es que el Presentador se usa en MVP para desacoplar la vista y el modelo.

Fue inventado en los años 90 bajo una iniciativa conjunta en C++ con Apple, IBM y HP considerando desarrollar una alternativa al patrón MVC. Su acrónimo de las palabras Model, View y Presenter, hacen referencia a los componentes esenciales de este patrón. Se han identiciado en implementaciones variantes como Supervising Controller y el Passive View.

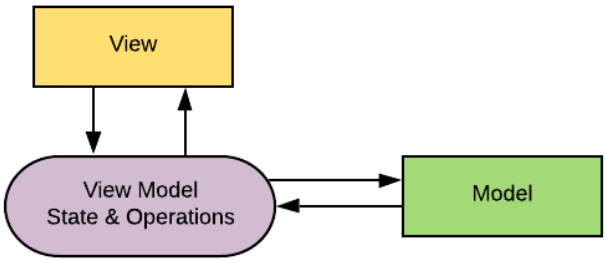
La diferencia entre MVP y otros patrones de UI es la independencia entre modelo y vista, ya que la vista no conoce que existe un modelo y el modelo no conoce que exista una vista. Su comunicación se logra a través del Presenter.

Este patrón nos aporta características peculiares como:

- El estado datos se maneja siempre a nivel de modelo, concretamente en la capa de servicio, cuando no en servidor.
- Se evitan cadenas de eventos.
- Minimiza llamadas al servidor, debido a que la lógica y los datos se entregan a nivel de servicio.
- Reducción en integración, cada pieza de software cuenta con sus responsabilidades y obligan a distribuir lógica entre modelo y vista, dejando de lado a los intermediarios.
- Favorece el uso de test unitarios.

MVVM (ModelView - ViewModel)

Un patrón joven, su creación fue 2005 John Gossman de Microsoft.



MVVM tiene como finalidad de tratar de desacoplar lo máximo posible la interfaz de usuario de la lógica de la aplicación. Se caracteriza por los siguientes elementos:

Modelo: Hace referencia a la capa de datos y a la lógica de negocio, se denomina también como el objeto del dominio. El modelo contiene la información, pero nunca las acciones o servicios que la manipulan. En ningún caso tiene dependencia alguna con la vista.

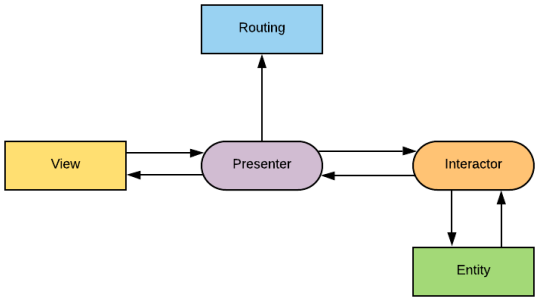
Vista: Su objetivo es representar información a través de los elementos visuales que la integran. Las vistas en MVVM son activas y dínamicas, contienen comportamientos, eventos y enlaces a datos que, en cierta manera, necesitan tener conocimiento del modelo subyacente. En Xamarin Forms podemos crear nuestras interfaces a través de código C# o XAML.

Modelo de vista (View Model): es un intermediario entre el modelo y la vista, contiene toda la lógica de presentación y se comporta como una abstracción de la interfaz. La comunicación entre la vista y el view model se realiza por medio los enlaces de datos.

Las responsabilidades de cada capa de MVVM son similares a las de MVP. ViewModel corresponde a la capa Presenter, excepto que hay más operaciones de enlace de datos en la capa de Vista de MVVM. Dado que está basado en MVVM, debe tener sus ventajas o ventajas sobre MVP, para decirlo sin rodeos, es compensar esas versiones cortas de MVP.

Comparando MVP y MVVM, puede ver que la vista del MVP activa la lógica empresarial de Presenter y la operación de Presenter para volver a llamar para cambiar la visualización de la Vista. El enlace de datos de MVVM se utiliza para realizar la lógica más claramente y el código es menor. Esta es la mejora de MVVM sobre MVP.

VIPER



VIPER dejó por un lado VC, colocándose de nuevo en capas, cada módulo se puede reutilizar y probar por separado. Sin embargo, ha generado una gran afectación de contar con demasiado código y una lógica complicada.

Ideal para grandes proyectos, para proyectos pequeños es ideal contar con tiempo suficiente debido a la dedicación que requiere la correcta separación de módulos. Es importante considerar los siguiente para el aprovechamiento de VIPER.

¿Cuándo usar VIPER?

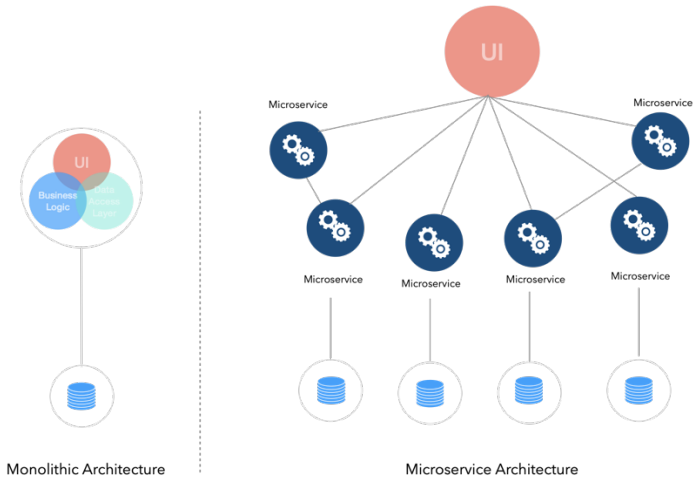
- VIPER se considera para proyectos que se visualicen en un escenario escalable, ya que permite trabajar a más de un desarrollador en la misma aplicación móvil.
- Su creación a través de módulos facilita la detección de errores, integración de nuevas funcionalidades, código ordenado y aprovechamiento de pruebas unitarias.

¿Cuándo no usar VIPER?

VIPER podría afectar a la estrategia de desarrollado, ya que al ser usado en proyectos pequeños, podría emplearse más tiempo en desarrollar la arquitectura.

Microservicios

La arquitectura de microservicios consta de una colección de **servicios autónomos** y pequeños, son **independientes** y cada uno implementa una funcionalidad de **negocio individual. Estos servicios se comunican entre sí mediante API's bien definidas.** Debido a que se ejecutan de forma autónoma, cada servicio se puede desarrollar, implementar y escalar sin afectar el funcionaiento de otros servicios.



¿Qué beneficios aportan los microservicios?

Versatilidad tecnológica: Con los microservicios podemos elegir la tecnología que mejor se adapte a las necesidades de cada servicio y al no seguir un enfoque de diseño único los equipos pueden tener la libertad de escoger las herramientas que mejor responda a necesidades específicas.

Agilidad: Debido a que los microservicios se implementan de forma independiente es más fácil hacer una actualización o corrección de un servicio sin tener que volver a implementar toda la aplicación, lo que significa que también acorta los tiempos de ciclo de desarrollo y aumenta el rendimiento de la organización.

Escalabilidad: Los microservicios **permiten que cada servicio puede ser escalado de forma independiente.** Los subsistemas que necesiten más recursos podrán ser escalados horizontalmente, sin tener que escalar toda la aplicación, lo cual permite a los equipos adecuarse a las necesidades de la infraestructura.

Aislamiento de errores: Los errores que puedan ser presentados no afecta la capacidad de otros servicios, ni interrumpe el buen funcionamiento de los demás, siempre y cuando los microservicios de nivel superior estén diseñados para controlar errores adecuadamente.

Rapidez: La arquitectura de microservicios permite que los procesos de la implementación y actualización de las aplicaciones sean más **agiles.**

Código reutilizable: El software, al ser dividido en módulos pequeños permite a los equipos usar funciones para diferentes propósitos, lo cual permite que los desarrolladores pueden crear nuevas capacidades, sin tener que escribir código desde el inicio.

¿Cuáles son los desafíos del uso de microservicios?

Debemos tomar en cuenta que una aplicación de microservicios puede ser más "complicado" de controlar ya que tiene más partes en movimiento y si contamos con un gran número de microservicios la **gestión e integració**n de los mismos se vuelve más compleja, al igual que **la realización de pruebas y tests globales**.

Otro de los grandes desafíos es la **integridad de datos**, puesto que cada microservicio es responsable de la conversación de sus propios datos y si no se gestiona correctamente esto podría traer algunos problemas.

A pesar de ello, cada inconveniente que surga durante el desarrollo o en la implementación puede resolverse, siempre y cuando cuentes con un **equipo de desarrolladores con un alto nivel de expertise.**

Serverless (FaaS)

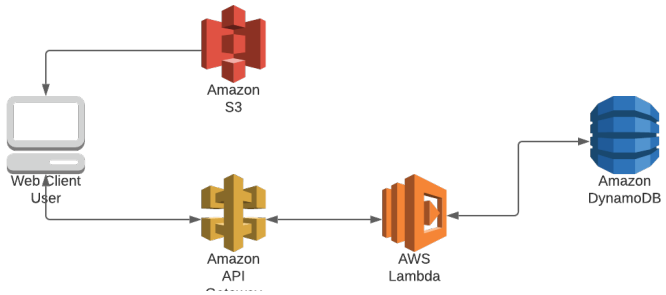
Serverless, tambien conocido como "Function as a Service", es un patrón de diseño de software en el que las aplicaciones son alojadas por un servicio de terceros, lo que elimina la necesidad de que el desarrollador deba administrar el software y el hardware del servidor. Las aplicaciones se dividen en funciones individuales que se pueden invocar y escalar individualmente.

Beneficios sobre Serverless

Mediante el uso de una arquitecturas serverless, los desarrolladores se pueden enfocar en el producto principal en lugar de preocuparse por la administración y el funcionamiento de los servidores, o los tiempos de ejecución, tanto en la nube como en las instalaciones. Gracias a esta reducción de gastos, los desarrolladores pueden emplear tiempo y energía en desarrollar productos increíbles que sean de confianza y que se puedan escalar.

Es una forma muy rentable de pagar los recursos de cloud e instancias. Permitiendo que las organizaciones, solo paguen por las veces que se llaman a sus funciones, en lugar de pagar para tener su aplicación siempre encendida y esperando solicitudes en tantas instancias diferentes.

A través de AWS puedes lograr arquitecturas serverless, aprovechando de Lambdas:



Sin duda, para compañías lideres como **Amazon, Spotify o Netflix**, está claro que Serverless son la mejor opción, ya que hacen más eficientes su operación y transacciones dentro de sus servicios.

Desde nuestra perspectiva la arquitectura Serveless se identifica como la ideal para innovar en productos de Software, ofreciendo alta disponibilidad, confiabilidad y rápidez a sus usuarios, es una arquitectura estratégica para orientar grandes retos tecnológicos.

El problema más recurrente en la mayoría de las compañías es la adopción y aprovechamiento de estás arquitecturas, puesto que es complicado lograr que los equipos se alineen para este tipo de arquitecturas, especialmente cuando se trata de grandes compañías que no están acostumbradas a la introducción de nuevos cambios o a trabajar de diferente manera.

Reducir el riesgo en las arquitecturas y contar con los servicios que las nubes nos ofrecen es la mejor manera de agilizar tu negocio. **La creación de software de alta disponibilidad es vital para competir durante la transformación digital.**

En **XID – Digital Services** nos especializamos en desarrollador software de alta disponibilidad y creemos que es indispensable para el crecimiento de tu negocio que conozcas las tecnologías más actualizadas para que tengas libertad de decisión.

Recuerda que tu creatividad no tiene límites.

¿Tienes algún proyecto en mente? Contáctanos en: Tell.Us@xid.com.mx

Fuentes:

- Architectural Pattern / Wikipedia
- Twilio / Serverless Architecture
- BBVA / Arquitecturas Serverless
- Software Crafters / Differences Between Patterns
- AWS / Serverless LAMBDA

¡Muchas gracias!
Hecho con❤️

#WeAreXID⚡💚💙💜